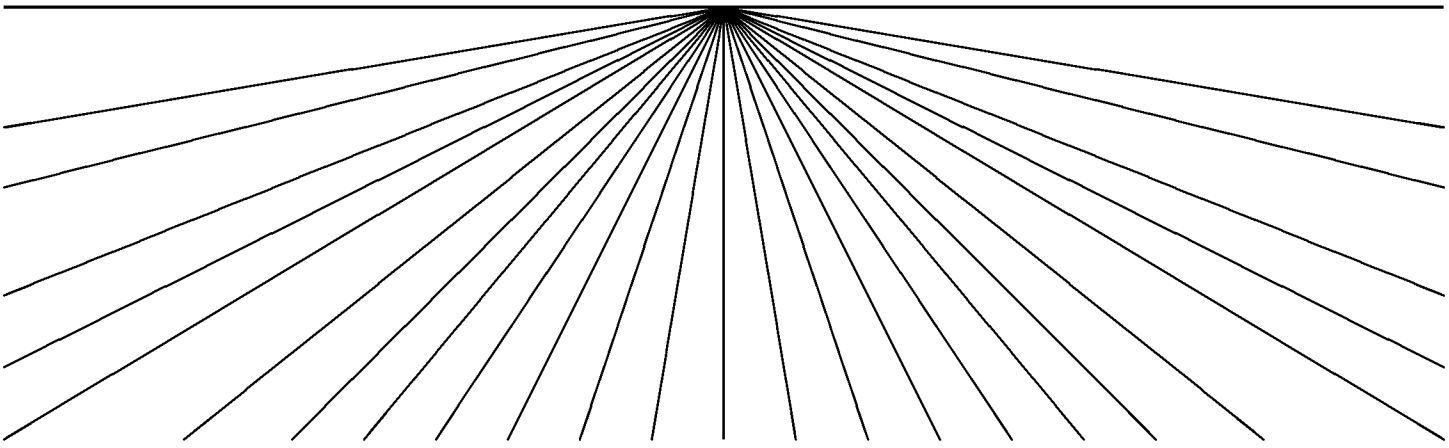
facultad de informática

universidad politécnica de madrid

Automatic Compile-Time Parallelization of CLP Programs by Analysis and Transformation to a Concurrent Constraint Language

Francisco Bueno
María José García de la Banda
Manuel Hermenegildo

TR Number CLIP3/95



Automatic Compile-Time Parallelization of CLP Programs by Analysis and Transformation to a Concurrent Constraint Language

---

Technical Report Number: CLIP3/95

June 1995

Keywords

(Constraint) Logic Programming, Concurrent Constraint Programming, Compile-time Analysis, Abstract Interpretation, Independent And-Parallelism, Program Transformation, Parallelization

Acknowledgements

This work was funded in part by ESPRIT project 7195 "ACCLAIM" and by CICYT projects TIC93-0975-CE and TIC93-0737-C02-01 IPL-D.

Abstract

The concept of independence has been recently generalized to the constraint logic programming (CLP) paradigm. Also, several abstract domains specifically designed for CLP languages, and whose information can be used to detect the generalized independence conditions, have been recently defined. As a result we are now in a position where automatic parallelization of CLP programs is feasible. In this paper we study the task of automatically parallelizing CLP programs based on such analyses, by transforming them to explicitly concurrent programs in our parallel CC platform (*CIAO*) as well as to AKL. We describe the analysis and transformation process, and study its efficiency, accuracy, and effectiveness in program parallelization. The information gathered by the analyzers is evaluated not only in terms of its accuracy, i.e. its ability to determine the actual dependencies among the program variables, but also of its effectiveness, measured in terms of code reduction in the resulting parallelized programs.

Given that only a few abstract domains have been already defined for CLP, and that none of them were specifically designed for dependency detection, the aim of the evaluation is not only to asses the effectiveness of the available domains, but also to study what additional information it would be desirable to infer, and what domains would be appropriate for further improving the parallelization process.

Contents

1 Introduction

Independence is the main parallelization principle in both the independent-and parallelism model [Con83, DeG84, Her86, WHD88, HG90] and the *DDAS* model [She92], and is included in several CC models recently proposed which combine these forms of parallelism [War90, GSCYH91, JH91] (in one of these models –the AKL model– independence forms an integral part of the "stability" principle) . The concept of independence refers to the conditions that the run-time behavior of the goals to be run in parallel must satisfy to guarantee the correctness and efficiency of the parallelization with respect to the sequential execution.

Automatic parallelization is thus closely related to the detection of some notion of independence and therefore involves an overhead. It is vital that such overhead remain reasonable. Several parallelization techniques have been defined for this purpose. A very effective approach, proposed initially by R. Warren et al [WHD88, HWD92] and developed further in [MH90b, CH94, BdlBH94a], is to combine local analysis and possibly some run-time checking with a highly sophisticated data-flow analysis based on the technique of abstract interpretation [CC77], aimed at reducing such run-time checking to a minimum.

In the context of traditional logic programming (LP), different independence conditions have been studied and proved correct [HR95, CH94]. Also, many sophisticated abstract domains relevant to the automatic parallelization task [Son86, CDY91, JL89, MH92, MH91, CMB$^+$93] have been proposed. The work presented in [BdlBH94b], reports on the integration of such domains in a second-generation analysis framework [MH92, MH90a], describes their integration into the complete parallelizing compiler and run-time system [HG91], and studies their efficiency, accuracy, and effectiveness in program parallelization, based on the independence conditions mentioned above. The experiments confirm the importance of global data-flow analysis in the parallelization task.

The concept of independence has been recently generalized to the constraint logic programming (CLP) paradigm [dlBHM93]. Also, several abstract domains specifically designed for CLP languages, and whose information can be used to detect the generalized independence conditions, have been recently defined [dlBH93, DJBC93, dlBHB$^+$95]. As a result we are now in a position where automatic parallelization of CLP programs is feasible.

In this paper we study the task of automatically parallelizing CLP programs based on such analyses, by transforming them to explicitly concurrent programs in our parallel CC platform (*CIAO*) as well as to AKL. Note that the translation into AKL that we

propose requires additional steps to those presented in this paper, which are described in an accompanying paper [BH95]. We describe the analysis and transformation process, and study its efficiency, accuracy, and effectiveness in program parallelization. The information gathered by the analyzers is evaluated not only in terms of its accuracy, i.e. its ability to determine the actual dependencies among the program variables, but also of its effectiveness, measured in terms of code reduction: analysis can help both in avoiding parallelizing expressions which can never succeed, and in reducing the conditions on other expressions.

Given that only a few abstract domains have been already defined for CLP, and that none of them were specifically designed for dependency detection, the aim of the evaluation is not only to asses the effectiveness of the available domains, but also to study what additional information it would be desirable to infer, and what domains would be appropriate for further improving the parallelization process.

The rest of the paper is structured as follows. Section 2 gives an brief description of the evaluation environment – i.e., the parallelizing compiler and the language. Section 3 discusses the generalization of independence to CLP, which is a non-trivial extension of that of LP. Based on this, Section 4 presents suitable parallelization conditions and tests for CLP programs. Section 5 describes the structure and task of the "annotators" – the actual parallelizers which interface with the analyzers – and the interface itself. Section 6 then presents the different domains and the framework they are constructed on, and discusses the usefulness of the information encoded by each domain for the annotation process in terms of the defined interface. Sections 7 and 8 describe the experiments and presents the results obtained from those experiments. Finally, Section 9 presents our conclusions.

2 The *CIAO* Compiler

The transformations that we propose for CLP parallelization are implemented in the context of the *CIAO* compiler [Bue95] and system [HtCg94, Bue95, CH95]. This compiler, whose overall structure is shown in Figure 2, is a program analysis and transformation workbench which can deal with several programming models simultaneously and perform several translations among them. The compiler includes a number of analysis modules, one of which is capable of analyzing LP and CLP programs using various domains, inferring information which is useful, among other things, for independence detection. It also includes a compile-time parallelization module, currently aimed at uncovering goal-level, restricted (i.e., fork and join) independent and-parallelism. The parallelization module is capable of parallelizing automatically source LP and CLP code directly, and in a user-transparent way (except for the increase in performance). This parallelization is performed as a source to source transformation, which is called an *annotation*. Only a subset of the *CIAO* language operators is needed in this translation, namely the `&/2` operator, which encodes the above mentioned type of parallelism. Compiler switches (implemented as "flags") determine whether or not code will be par-

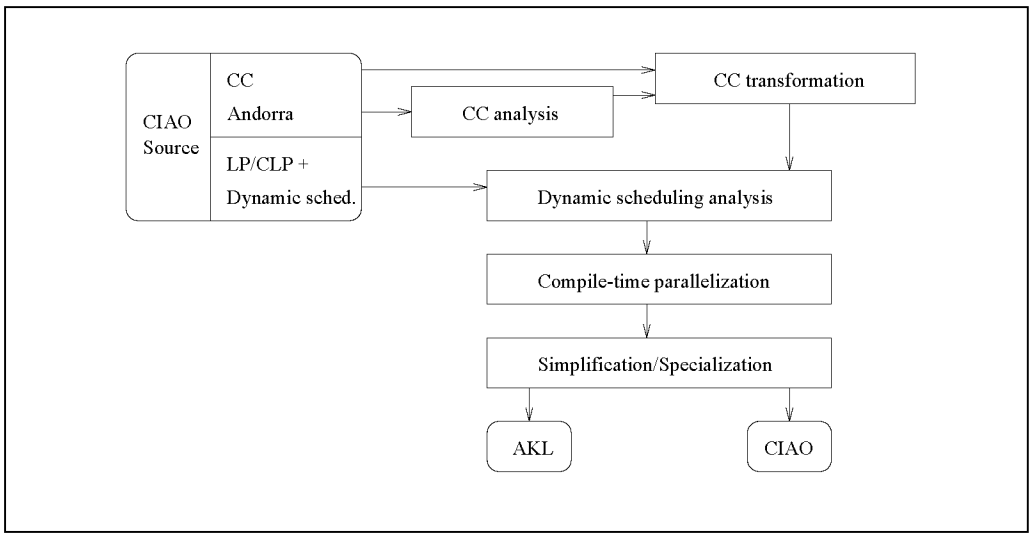


Figure 1: Evaluation System

allelized and through which type of analysis. Alternatively, concurrent code can always be written by the user (the compiler then checking such code for correctness).

The two modules mentioned above will be the ones used in the evaluation of the techniques presented in this paper. Additionally the compiler has side-effect and granularity analyzers (not depicted in the figure), which infer information which can yield the sequentialization of goals (even when they are independent) based on efficiency or maintenance of observable behavior. Furthermore, other modules bridge the semantic gaps between the different programming paradigms supported. In fact, the compiler is also capable of parallelizing more general concurrent or dynamically scheduled code, sometimes after some transformations.

The target of the compiler is the *CIAO* system, a run-time system capable of exploiting this and other forms of parallelism using one or more processors, in shared-memory or distributed execution. Alternatively, as mentioned before, AKL can be used as a target, using the techniques described in [BH95].

3 Independence in CLP

As mentioned in the introduction, independence has proved to be a very useful concept in the context of conventional logic programming, as it is the base of many optimizations. Among all, we focus in parallel and concurrent execution models, where different notions of independence [HR89, HR90] (strongly related to the concept of "stability" [HJ90]) are often used to ensure that the parallel or concurrent execution of a set of goals in the body of a clause is correct and efficient w.r.t. a given sequential execution [Con83, DeG84, HG90, JH91, War90].

The general, intuitive notion of independence that we would like to characterize can be expressed as follows: a goal `q` is independent of a goal `p` if `p` does not "affect" `q`. A goal `p` is understood to affect another goal `q` if `p` changes the execution of `q` in an "observable" way. Observables include changing the solutions that `q` produces and also changing the time that it takes to compute such solutions. This time can change either because the actual number of reduction steps differs and/or because the amount of work involved in performing each of those steps differs in a significant way.

Previous work in the context of traditional logic programming languages [HR89, HR90] has concentrated on defining independence in terms of *preservation of search space*: if the execution of a goal `p` does not modify the search space of the goal `q` then neither the solutions produced by `q` nor the number of reduction steps will differ regardless of the execution of `p` (we will address below the issue of changing the amount of work performed at each reduction step). This idea has been formalized by first defining two basic notions of independence —*strict* and *non-strict independence*— and then showing that they are in fact sufficient for guaranteeing preservation of search space.

It could be thought that the traditional concepts of independence might carry over trivially to CLP. However, the constraint systems used and their solvers can behave in ways that are quite different in fundamental aspects (from the point of view of independence) from the behavior of equalities over first-order terms using the standard unification algorithm, which was the situation assumed for the LP notions. Thus, the basic results for LP do not hold in general for CLP and new notions of independence are needed in order to ensure search space preservation.

In [dlBHM93, dlBG94] several notions of independence for CLP are presented, each one useful for a class of applications, and several results on the implications of these definitions shown. Conditions which can be detected at compile-time and/or run-time and which are proved to be sufficient for all these definitions are also presented.

In the next section we discuss those notions which are useful for automatic parallelization. We will assume knowledge regarding both the abstract interpretation technique and the constraint logic programming paradigm (see [CC77, JM94]). The notation used in the rest of the paper is as follows. Upper case letters generally denote collections of objects, while lower case letters generally denote individual objects. u, v, w, x, y, z will denote variables, p, q will denote predicate symbols, c will denote a constraint, π will denote the constraint store, and g will denote a goal. These symbols may be subscripted or have an over-tilde. $\vec{x}$ denotes a sequence of distinct variables. $\exists_{-\vec{x}}\phi$ denotes the existential closure of the formula ϕ except for the variables $\vec{x}$. $\vec{\exists}\phi$ denotes the full existential closure of the formula ϕ.

4 Parallelization Conditions and Tests

As mentioned in the introduction, the concept of independence refers to the conditions that the run-time behavior of the goals to be run in parallel must satisfy to guarantee the correctness and efficiency of the parallelization with respect to the sequential execution. Consider the goals $g_1(\bar{x})$ and $g_2(\bar{y})$ and the store π. As shown in [dlBHM93], the parallelization of $g_1(\bar{x})$ and $g_2(\bar{y})$ w.r.t. π is correct and efficient (for LP) iff for any answer c derivable from the state $(g_1(\bar{x}), \pi)$, the search space of the states $(g_2(\bar{y}), \pi)$ and $(g_2(\bar{y}), c)$ are the same. This in turn is satisfied iff for any answer derivable from the state $(g_1(\bar{x}), \pi)$ and for any partial answer derivable from the state $(g_2(\bar{y}), \pi)$ we have that they are consistent. Then we say that $g_2(\bar{y})$ is *strongly independent* from $g_1(\bar{x})$ w.r.t. π. Note that, as opposed to the conditions previously defined in the literature, strong independence is not only a sufficient but also a necessary condition for ensuring both efficiency and correctness of the parallel execution.

When parallelization might be performed in a shared-memory machine (as can happen in our system) a symmetric notion is needed: the parallelization of $g_1(\bar{x})$ and $g_2(\bar{y})$ w.r.t. π in a shared environment is correct and efficient (for LP) iff for any partial answer c derivable from the state $(g_1(\bar{x}), \pi)$, the search space of the states $(g_2(\bar{y}), \pi)$ and $(g_2(\bar{y}), c)$ are the same, and vice versa. This in turn is satisfied iff for any partial answer derivable from the state $(g_1(\bar{x}), \pi)$ and for any partial answers derivable from the state $(g_2(\bar{y}), \pi)$, we have that they are consistent. We then say that $g_1(\bar{x})$ and $g_2(\bar{y})$ are *search independent* w.r.t. π.

Search independence cannot always be checked at run-time without actually executing the goals, i.e. it is not decidable *a priori*. This is understandable since this notion depends on the run-time behavior of the goals. However, search independence can be checked at compile-time by analyzing the target program for an abstract domain capable of ensuring the consistency and inconsistency of some abstractions, such as the abstract domains LSign [MS94] or Free [DJBC93]. Therefore, let $\alpha(\pi)$ be the abstraction of the store π in such a domain, and let AC_1, AC_2 be the success patterns inferred from the analysis of $g_1(\bar{x})$ and $g_2(\bar{y})$, respectively, with calling pattern $\alpha(\pi)$. If the abstraction resulting from the abstract conjunction of AC_1 and AC_2 can be ensured to be satisfiable, $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent w.r.t. π. Therefore, they can be executed in parallel, and do so without requiring any run-time independence / stability test. Note that for correctly detecting search independence, the information obtained during the analysis of any failure branch has to be included in the success pattern of the goal.

Unfortunately, as shown in [dlBHM93], search space preservation (or search independence) is not sufficient for ensuring the efficiency of the parallelization when arbitrary CLP solvers are taken into account. The reason is that while the number of reduction steps will certainly be constant if the search space is preserved, the cost of each step will not: modifying the order in which a sequence of primitive constraints is added to the store may have a critical influence on the time spent by the constraint solver algorithm

in obtaining the answer, even if the resulting constraint is consistent (in fact, this issue is the core of the reordering application described in [MS92]). Thus, optimizations which vary the intended execution order established by the user, such as parallel or concurrent execution, may actually yield a slow-down even if search space preservation is ensured. This issue of the variance of the cost of adding primitive constraints to the store has been ignored as a factor of negligible influence in traditional logic programming languages, due to the specific characteristics of the standard unification algorithms. However, in constraint logic programs we must also consider an orthogonal issue – *independence of constraint solving* – which characterizes the properties of the constraint solver behavior when changing the order in which primitive constraints are considered.

In order to take this issue into account, let us introduce a new concept [dlBHM93]. Let $def\_vars(c)$ denote the set of definite variables in the constraint c (i.e., the set of variables uniquely defined by c). Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint store π iff

$$(\bar{x} \cap \bar{y} \subseteq def\_vars(\pi)) \; and \; (\exists_{-\bar{x}}\pi \wedge \exists_{-\bar{y}}\pi \rightarrow \exists_{-\bar{y}\cup\bar{x}}\pi)$$

In words, if all shared variables are definite, and the constraints on the variables of both goals are implied by the conjunction of the constraints on the variables of each goal.[1]

This same definition can also be applied to terms and constraints without any change. Note that if a term (or a goal) is ground, then it is projection independent from any other term (or goal). Also, note that projection independence is symmetric, but not transitive.

The above notion is important for three reasons [dlBHM93]. First, most solvers are *projection independent*, i.e. are not significantly affected by the order in which primitive constraints are added to store π if such constraints are projection independent w.r.t. π. Second, if goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for a constraint π, for any two constraints c_1, c_2 added by $g_1(\bar{x})$ and $g_2(\bar{y})$, respectively, c_1 and c_2 are projection independent w.r.t. π, and therefore the parallelization of the goals will not affect the solver behavior. And third, if $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for π, they are also search independent for π. As a result, for this particular kind of solvers, the parallelization of two goals in store π can be ensured to be correct and efficient if such goals are projection independent w.r.t. π. Finally, note that projection independence is an a priori notion and thus cannot only be detected at compile-time but also at run-time.

Unfortunately, the cost of performing a precise projection at run-time might be too high. A pragmatic solution is, whenever projection independent cannot be ensured at compile-time, to simplify the run-time independence / stability tests by just checking if the variables involved are 'linked" through the primitive constraints in the constraint

[1] Note that $(\exists_{-\bar{x}}\pi \wedge \exists_{-\bar{y}}\pi \leftarrow \exists_{-\bar{y}\cup\bar{x}}\pi)$ is always satisfied.

store. More formally, let Π denote a sequence of constraints. The relation $link_\Pi(x, y)$ holds for variables x and y if there is a primitive constraint c in Π such that $\{x, y\} \subseteq vars(c) \setminus def\_vars(\Pi)$. The relation $links_\Pi(x, y)$ is the transitive closure of $link_\Pi(x, y)$. We lift *links* to sets of variables by defining $Links_\Pi(\bar{x}, \bar{y})$ iff $\exists x \in \bar{x}$ and $\exists y \in \bar{y}$ such that $links_\Pi(x, y)$.

Then, we can ensure that goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint Π if $\neg Links_\Pi(\bar{x}, \bar{y})$. Note that this property does not depend on the syntactic representation we choose for Π. In fact if the solver keeps a "normal form" for the current constraints we are better off using the normal form rather than the original sequence of constraints as this allows the definition to be simplified. More precisely: constraints Π are in *normal form* if they have form:

$$x_1 = f_1(\bar{y}) \wedge x_2 = f_2(\bar{y}) \wedge \ldots \wedge x_n = f_n(\bar{y}) \wedge \Pi'$$

where the x_i are distinct and disjoint from the variables $\bar{y}$ and $vars(\Pi') \subseteq \bar{y}$. Associated with the normal form is an assignment ψ to the eliminated variables, namely,

$$[x_1 \mapsto f_1(\bar{y}), \ldots, x_n \mapsto f_n(\bar{y})]$$

It is straightforward to verify that $Links_\Pi(x, y)$ iff $Links_{\Pi'}(vars(\psi(x)), vars(\psi(y)))$.

It is interesting to note that despite the fact that we initially considered a left-to-right execution rule, the a priori conditions given in this section are valid independently of any computation rule. This is due to the fact that these conditions are defined in terms of the information provided by the constraint store readily available before executing the goals. Thus, the conditions will remain valid no matter which computation rule will be later applied in the execution of the goals. Therefore, the results obtained in this section can be directly applied to non-deterministic constraint languages with other computation rules, such as AKL [JH91] (or, in general, any non-deterministic concurrent constraint language [Sar89]).

5 The Parallelization Process Based on a priori Notions

Several approaches to automatic parallelization have been proposed in the literature. The solution used in the *CIAO* parallelizing compiler is, as mentioned before, that proposed initially by R. Warren et al [WHD88] and developed further in [MH91, MH89b, MH90b, MH92]: combining local analysis and possible run-time independence/stability checking with a highly sophisticated data-flow analysis based on the technique of abstract interpretation [CC77], which reduces such checking. In this section we will focus on automatic parallelization based on notions of independence which are decidable at run-time.

In this context, the automatic parallelization process is performed as follows. Firstly, if required by the user, the CLP program is analyzed using one or more global analyzers, aimed at inferring useful information for detecting independence. Secondly, since

side-effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [MH89a] which propagates the side-effect characteristics of builtins determining the scope of side-effects. Finally, the *annotators* perform a source-to-source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before. Additionally, while annotating each clause, the annotators can also invoke local analyzers in order to infer further information regarding the literals in the clause. In the current implementation, side-effect builtins and procedures are not parallelized. Also, some limited knowledge about the granularity of the goals, in particular the builtins, is used by means of a local analysis. As a result, the only kind of builtins allowed to be run in parallel are the meta–calls.

The annotation process is divided into two subtasks. The first is concerned with identifying the dependencies between each two goals in a clause and generating the minimum number of tests which, when (a priori) evaluated at run-time, ensure their independence. The second task is concerned with the core of the annotation process, namely, application of a particular strategy to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step, hopefully further optimizing the number of tests.

5.1 Identifying Dependencies

The dependencies between goals can be represented as a dependency graph [Con83, JL88, Kal87, Lin88, CDD85, MH90b]. Informally, a dependency graph is a directed acyclic graph where each node represents a goal and each edge represents in some way the dependency between the connected goals. A conditional dependency graph (CDG) is one in which the edges are adorned with independence conditions. If those conditions are satisfied, the dependency does not hold. In an unconditional dependency graph (UDG) dependencies always hold, i.e., conditions are always "false."

Given a collection of literals, we would then like to be able to generate at compile-time a condition *i_cond* which, when evaluated at run-time, would guarantee that the goals which are instantiations of such literals are independent for the particular notion used. Furthermore, we would like that condition to be as efficient as possible, hopefully being more economical than the explicit application of the condition. Consider the set of conditions which includes "true", "false", or any set, interpreted as a conjunction, of one or more of the following tests: $def(x)$, $indep(x, y)$ where x and y can be goals, variables, or terms in general. Let $def(x)$ be true when x is definite, i.e. constrained to a unique value ("ground" in the Herbrand domain), and false otherwise. Let $indep(x, y)$ be true when x and y are independent for the particular notion applied and false otherwise. In the case of projection independence, the test can be implemented by either projecting the store over the two variables and then checking that there is no constraint involving both variables, or by traversing the original store to detect the

(less expensive) *Link* condition defined in the previous section.

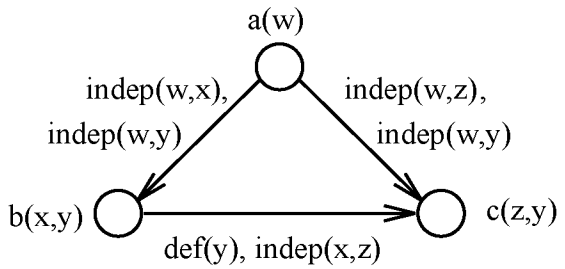
Consider the literals $g_1, \ldots, g_n$. If no global information is provided, an example of such a correct *i_cond* is $\{def(x) | \forall x \in SVG\} \cup \{indep(x,y) | \forall (x,y) \in SVI\}$, where SVG and SVI are defined as follows:

- $SVG = \{v \mid \exists i,j, i \neq j$ with $v \in vars(g_i) \cap vars(g_j)\}$;
- $SVI = \{(v,w) \mid v,w \notin SVG$ and $\exists i,j, i < j$ with $v \in vars(g_i)$ and $w \in vars(g_j)\}$.

If the above condition is satisfied the literals are independent for every possible constraint store, thus ensuring that the goals resulting from the instantiations of such literals will also be independent.

It is easy to see that in general a definiteness check is less expensive than an independence check, thus a condition, such as the one given, where some independence checks are replaced by definiteness checks is obviously preferable. The left-to-right precedence relation for the literals in the clause above can be represented using a directed, acyclic graph in which we associate with each edge which connects a pair of literals, the tests needed for ensuring their independence.

Example 1 *Consider the following literals in a program clause:* `..., a(w), b(x,y), c(z,y), ....` *The associated dependency graph is illustrated in Figure 1.* □



5.2 Simplifying Dependencies

The annotation process can be improved by using compile–time information obtained either by the user or by local or global analysis. This improvement is based on identifying tests which are ensured to either fail or succeed w.r.t. the information available: if a test is guaranteed to succeed, it can be reduced to true, thus eliminating the edge; if a test is guaranteed to fail, it can be reduced to false, yielding an unconditional edge.

In order to do this, and for each clause C, we translate the compile-time information associated to each program point i in C, into a *domain of interpretation GI* for definiteness and independence: a subset of the first order logical theory, such that each element κ of GI defined over the variables in C is a set of formulae (interpreted as their

conjunction) containing only predicates of $def(x)$ and $indep(y,z), \{x,y,z\} \subseteq vars(C)$, and such that $\forall \kappa \in GI\ \kappa \not\vdash false$, and:

$$\begin{array}{lll} \kappa & \supseteq & \{def(x) \rightarrow indep(x,y) | \{x,y\} \subseteq vars(C)\} \cup \\ & & \{def(x) \leftrightarrow indep(x,x) | x \in vars(C)\} \end{array}$$

For the sake of simplicity, in the rest of the paper this formula will be assumed to be part of any κ, although not explicitly written down. Thus, any time we write $\kappa = K$, κ should be interpreted as K augmented with the above set.

For any program point i of a clause C where a test T_i on the definiteness and independence of the clause variables is checked, the simplification of such test, based on an element $\kappa_i \in GI$ over the variables of C, is defined as the refinement of T_i to yield $T_i' = improve(T_i, \kappa_i)$, where:

$$improve(T_i, \kappa_i) = \left\{ \begin{array}{llll} if & \exists t \in T_i \ s.t.\ \kappa_i \vdash \neg t & then & false \\ elseif & \kappa_i \vdash T_i & then & true \\ else & for\ some\ t \in T_i & \{t\} \cup improve(T_i \setminus \{t\}, \kappa_i \cup \{t\}) & \end{array} \right.$$

Note that there is an implicit restriction on the selection of $t \in T_i$ in the above definition of *improve*: the order in which t is selected can influence the result of *improve*. Consider $\kappa_i = \{def(x) \rightarrow def(y)\}$ and $T_i = \{def(x), def(y)\}$. By selecting first $def(y)$ the final result is $T_i' = T_i = \{def(x), def(y)\}$, whereas by selecting first $def(x)$ the final result is $T_i' = \{def(x)\} \subset T_i$, which is simpler. We will avoid such non deterministic behavior by first selecting definiteness conditions (because of their lower cost at run–time), then those which does not appear as consequent in any atomic formula of κ_i, and then the rest. This will be done following a left-to-right selection rule.

The accuracy and the size (the number of atomic formulae for simple facts) of each κ depend on the kind of program analysis performed. In the next section we will explain how to build this formula from the domains of analysis used in our experiments.

5.3 Building Parallel Expressions

Given a clause, several annotations are possible. The second step in the annotation process aims at obtaining an optimal parallel expression among all the possibilities detected in the previous step by applying a particular strategy, hopefully further optimizing the number of tests. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause.

Example 2 *Consider again the sequence of literals* `a(w), b(x,y), c(z,y)` *in 1. A possible parallel expression for these goals would be:*

```
a(w), (def(y),indep(x,z) => b(x,y) & c(z,y))
```

An alternative would be:

```
(indep(w,x),indep(w,z),indep(x,z),def(y) => a(w) & b(x,y) & c(z,y))
```

and yet another alternative:

```
(indep(w,[x,y]) -> (a(w) & b(x,y), c(z,y))
                 ; a(w), (def(y),indep(x,z) => b(x,y) & c(z,y)))
```

and so on. □

For syntactic convenience we have used an additional construct `=>` usually known as the *Conditional Graph Expression (CGE)*. A CGE has the general form (*cond* `=>` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N$) where *cond* is a sufficient condition for running all $goal_i$ in parallel under the appropriate notion of independence. In the following, though, we will refer to CGEs in general as meaning "parallel expression."

Four different heuristic algorithms are implemented in the *CIAO* compiler, namely **CDG**, **UDG**, **MEL** [MH90b], and **URLP** [CH94][2]. The **CDG** algorithm seeks to maximize the amount of parallelism available in a clause, without being concerned with the size of the resultant parallel expression. In doing this, the annotator may switch the positions of independent goals. **UDG** does essentially the same as **CDG** except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel. **URLP** starts with the sequence of goals of the body of the clause and recursively tries to parallelize pairs of consecutive goals or groups of goals by a few simple rewriting rules, based on the dependencies found. **MEL** tries to find points in the body where it can be split into different parallel expressions (i.e., where edges labeled "false" appear) without changing the order given by the original clause and without building nested parallel expressions. At such points the clause body is broken into two, a CGE is built for the right part of the split sequence, and the process continues with the left part. In the following we will focus on the **MEL** algorithm, in the particular context of strict independence.

Once an expression has been built, it can possibly be further simplified, unless it is unconditional. E.g. the overall condition of the expression built by the annotation algorithm can possibly be further reduced, again based on the local or global information.

[2] **CDG** stands for Conditional Dependency Graph, **UDG** stands for Unconditional Dependency Graph, **MEL** stands for Maximal Expression Length, and **URLP** stands for Unconditional Recursive Linear Parallelization.

6 Global Analysis-Based Test Simplification

The analyzers we have studied include the Def domain [dlBH93]; the Free domain [DJBC93]; and the FD domain [dlBHB+95]. They have all been embedded in PLAI [MH92, MH90a], one of the components of the *CIAO* system compiler. PLAI is a domain independent analysis framework implemented in Prolog, based on the model of Bruynooghe [Bru91] with the optimizations described in [MH92, MH90a] and extended to CLP languages as defined in [dlBH93].

In this section we briefly introduce the definition of each domain and its concretization function, as well as its ability to capture the information needed to simplify the conditions in the parallel expressions.

6.1 Def Domain

The domain Def was defined for inferring *definiteness* information. Let $\wp(S)$ denote the powerset of a set S and $\wp_\emptyset(S)$ denote $\wp(S) \setminus \{\emptyset\}$. An abstract constraint $AC = (D, R)$ of the abstract domain Def is an element of $\wp(Pvar) \times \wp(Pvar \times \wp_\emptyset(\wp_\emptyset(Pvar)))$ satisfying the conditions imposed by the associated solved form. Intuitively, D approximates the set of variables which are known to be definite, and each element $(x, SS) \in R$ approximates definite dependencies which are known to hold between x and the sets of variables in SS. Such dependencies will allow the domain to perform accurate definiteness propagation.

The objective of the solved form is both to obtain a more compact representation and to reduce the cost of key operations. The idea is to propagate the definiteness information until (a) all variables known to be definite appear in D, and (b) for any $(x, SS) \in R$, SS is the *minimum* set of sets containing *all* possible definiteness relationships affecting x. For a more detailed definition of the solved form see [DJBC93].

The concretization function, γ : Def $\rightarrow 2^{Con}$, is defined for an abstract constraint $(D, R) \in$ Def as follows: $\gamma(D, R)$ approximates all concrete constraints π such that for every $x \in D : x \in def\_vars(\pi)$ and for every $(x, SS) \in R$, and every $S \in SS : S \subseteq def\_vars(\pi) \Rightarrow s \in def\_vars(\pi)$.

Let us now present the relation between Def and the domain GI. Consider an abstract constraint $AC_i = (D, R) \in$ Def for program point i of a clause C. The contents of the corresponding $\kappa_i \in GI$ are as follows:

- $def(x)$ if $x \in D$
- $def(x_1) \wedge \ldots \wedge def(x_n) \rightarrow def(y)$ if $(y, SS) \in R, \{x_1, \ldots, x_n\} \in SS$

Note that in this case κ_i contains neither $\neg def(x)$ nor $\neg indep(x, y)$ for any $\{x, y\} \subseteq vars(C)$, thus no tests in the CDG can ever be reduced to false with only this information.

Example 3 *Consider a clause C such that $vars(C) = \{x, y, z, v, w\}$ and an abstract constraint $AC = (\{x\}, \{(z, \{\{w\}, \{v\}\})\})$. The corresponding κ will be: $\{def(x), def(w) \rightarrow def(z), def(v) \rightarrow def(z)\}$.*

6.2 Free Domain

The domain Free aims at inferring whether variables are *free*, i.e. whether they can range over the whole domain specified by their type. For example, a variable that is constrained to be numerical but still ranges over the complete domain of numbers is considered as free. An abstract constraint AC of the abstract domain Free is an element of $\wp(\wp_\emptyset(Pvar))$ satisfying the conditions imposed by the associated normal form. Intuitively, AC approximates *possible* dependencies among variables, each set S in AC approximating the possible existence of a constraint affecting the variables in S. Such dependencies will allow the domain to take care of non-freeness propagation.

As shown in [DJBC93] sets that can be obtained as union of others (e.g. the set $\{x, y, z\}$ can be obtained, among others, from $\{x, y\}$ and $\{y, z\}$) are redundant with respect to non-freeness propagation. The associated normal form requires those redundant sets to be eliminated, thus obtaining a more compact representation without losing information useful for an accurate non-freeness propagation. For a more detailed definition of the normal form see [DJBC93].

Let $free(\pi)$ and $typed\_free(\pi)$ represent the set of variables which are, respectively, free to take any possible value (functor, arithmetic, etc) and free to take any value within a particular type, w.r.t. the concrete constraint π. The concretization function, γ : Free $\rightarrow 2^{Con}$, is defined for an abstract constraint $AC \in$ Free as follows: γAC approximates all concrete constraints π such that for every $x \in$ PVar: if $x \notin vars(AC) \Rightarrow x \in free(\pi)$ and if $\{x\} \notin AC \Rightarrow x \in typed\_free(\pi)$; and for every $S_1, \ldots, S_n \in AC$: there might be a constraint defined over the variables in $S_1 \cup \ldots \cup S_n$.

Let us now present the relation between Def and the domain GI. Consider an abstract constraint $AC_i \in$ Free for program point i of a clause C. The contents of the corresponding $\kappa_i \in GI$ are as follows:

- $indep(x, y)$ if $x \notin vars(AC_i)$
- $\neg def(x)$ if $x \notin vars(AC_i)$ or $\{x\} \notin AC_i$
- $indep(x_1, y_1) \wedge \ldots \wedge indep(x_n, y_n) \rightarrow indep(w, z)$ if $\forall S \in AC_i$: if $z \in S$ then $\exists j \in [1, n],\ \{x_j, y_j\} \subseteq S$

The first rule states that variables x and y are independent if x is free, i.e., it does not appear in the abstraction. Note that due to the normal form, as soon as a variable appears in at least on set of the abstraction, it can be possibly dependent of any other variables appearing in the abstraction. The second rule states that a variable x is non-definite if it is either free or typed-free. Finally, variables z and w are independent

if each set S in which z appears is eliminated due to the knowledge that variables x_i and y_i in S are independent, i.e., if w is known to be typed-free and, due to the knew knowledge, we can now ensure that it is not only typed-free, but free (since w disappears of the abstraction).

Also, note that for the Free domain κ_i contains $\neg def(x)$ thanks to the freeness information. However, it still cannot contain $\neg indep(x, y)$ for any $\{x, y\} \subseteq vars(C)$. Note that definiteness (and therefore definiteness implications) cannot be derived by an abstract constraint in Free.

Example 4 *Consider a clause C such that $vars(C) = \{x, y, z, v, w\}$ and an abstract constraint $AC = \{\{x\}, \{z\}, \{v\}, \{z, w\}\}$. The corresponding κ will be: $\{\neg def(y), \neg def(w), indep(y, x), indep(y, z), indep(y, v), indep(y, w), indep(z, w) \rightarrow indep(w, x), indep(z, w) \rightarrow indep(w, v)\}$.*

6.3 FD Domain

We have also considered the evaluation of the analyzer resulting from the combination of the Def and Free domains, described in [dlBHB+95] as the FD domain. The information approximated by this domain can be used to simplify the CDG simply by translating the information inferred by each domain into the GI domain, conjoining the resulting κs, and applying the techniques described in previous sections.

7 Experimental Results: Traditional Logic Programs

In this section we present and discuss a set of experiments aimed at evaluating the efficiency and accuracy of the three analyzers introduced in the previous section when dealing with traditional logic programs, w.r.t. that achieved by analysers which were specifically defined for LP and proved effective in the parallelization task. For this purpose, we will use the ShFr analyser [MH91], which was already integrated in the PLAI framework. This analyzer implements a sophisticated abstract domain defined to infer *groundness*, *set sharing*, and *freeness* information. The information inferred by this analyzer has been already evaluated in the parallelization of LP programs [BdlBH94b]. The results of that study show that this analyzer is very effective in the automatic parallelization task.

7.1 Benchmarks

The set of benchmarks used in this study are those used in the evaluation of the effectiveness of analysers defined for traditional LP languages for the automatic parallelization of LP programs [BdlBH94b]. These benchmarks range from very simple (toy) programs to real application programs. Table 1 gives good insight into their complexity useful for the interpretation of the results:

- AgV, MV are respectively the average and maximum number of variables in each clause analyzed (dead code is not considered);
- Cl, Ls, and Ps are, respectively, the total number of clauses, literals in the body of a clause, and predicates analyzed;
- Non, Sim, and Mut are respectively the percentage of predicates non–recursive, simply recursive and mutually recursive;
- Gs is the total number of different goals solved in analyzing the program, i.e., the total number of syntactically different calls.

Bench.	AgV	MV	Cl	Ls	Ps	Non	Sim	Mut	Gs
aiakl	4.07	9	14	23	7	43	57	0	9
ann	3.17	14	215	270	65	43	20	37	96
bid	2.20	7	49	57	19	68	32	0	28
boyer	2.36	7	141	59	26	73	4	23	49
browse	2.63	5	19	30	8	12	62	25	9
deriv	3.70	5	10	22	1	0	100	0	1
fib	2.00	6	3	6	1	0	100	0	1
grammar	2.13	6	15	8	6	100	0	0	7
hanoiapp	4.25	9	4	7	2	0	100	0	3
mmatrix	5.33	9	6	20	3	0	100	0	3
occur	3.12	6	8	11	4	25	75	0	4
peephole	3.15	7	155	132	26	46	8	46	30
progeom	3.59	9	17	22	9	33	67	0	13
qplan	3.18	16	152	237	46	39	33	28	54
qsortapp	3.29	7	7	9	3	0	100	0	4
query	0.19	6	52	9	4	100	0	0	4
rdtok	3.07	7	67	173	22	32	27	41	30
read	4.05	13	91	197	24	54	12	33	47
serialize	4.18	7	11	20	5	20	80	0	7
tak	7.00	10	2	10	1	0	100	0	1
warplan	2.47	7	94	122	29	52	31	17	103
zebra	2.06	25	18	23	6	67	33	0	10

Table 1: Benchmark Profiles: LP

The number of variables in a clause affects the complexity of the analysis because the abstract functions greatly depend on the number of variables involved. Note that when abstract unification is performed, the variables of both the subgoal and the head of the clause to be unified have to be considered. Therefore, the number of variables involved in an abstract unification can be greater than the maximum number of variables shown in the table. The number of recursive predicates affects the complexity of the fixpoint algorithm possibly increasing the number of iterations needed.

7.2 Efficiency

Table 2 presents the efficiency results in terms of analysis times in seconds (SparcStation 10, one processor, SICStus 2.1 #5, native code). The table shows for each benchmark and analyzer the average times out of ten executions. The last row shows the average time for each analyzer.

Benchmark	Def	Free	FD	ShFr
aiakl	2.75	1.01	3.84	0.15
ann	2.46	3.13	6.56	5.79
bid	0.14	0.54	0.28	0.22
boyer	1.41	2.11	4.19	2.62
browse	0.06	0.19	0.14	0.10
deriv	0.04	1.63	0.14	0.06
fib	0.01	0.02	0.03	0.01
grammar	0.07	0.10	0.17	0.07
hanoiapp	0.02	0.16	0.06	0.03
mmatrix	0.02	0.06	0.05	0.03
occur	0.02	0.16	0.05	0.04
peephole	1.03	4.55	4.27	2.24
progeom	0.08	0.39	0.15	0.12
qplan	0.57	5.10	1.27	1.02
qsortapp	0.03	0.14	0.06	0.04
query	0.03	0.07	0.08	0.04
rdtok	0.61	1.98	1.94	1.53
read	0.93	4.36	1.66	1.44
serialize	0.38	0.54	0.79	0.37
tak	0.01	0.05	0.03	0.02
warplan	2.18	0.99	4.62	5.01
zebra	0.1	0.3	0.2	0.1
Average	0.61	1.30	1.45	1.00

Table 2: Analysis Times: LP

Table 3 presents the efficiency results in terms of annotation times in seconds. The benchmarks have been parallelized with the **MEL** annotator in two different situations: with only the information provided by local analysis (Local in the table), and with that provided by both local analysis and one of the above mentioned global analyzers. The last row shows the average time for each analyzer.

7.3 Effectiveness

One way to measure the accuracy and effectiveness of the information provided by abstract interpretation-based analyzers is to count the number of CGEs which actually result in parallelism, the number of these which are unconditional, and the number of definiteness (groundness) and independence tests in the remaining CGEs, which

Benchmark	Local	Def	Free	FD	ShFr
aiakl	0.09	0.14	0.15	0.15	0.18
ann	0.78	0.99	1.24	1.36	4.07
bid	0.15	0.35	0.28	0.26	0.34
boyer	0.15	0.20	0.21	0.28	0.57
browse	0.08	0.11	0.12	0.13	0.15
deriv	0.07	0.07	0.08	0.07	0.09
fib	0.04	0.04	0.04	0.05	0.06
grammar	0.05	0.06	0.05	0.06	0.08
hanoiapp	0.06	0.05	0.06	0.07	0.06
mmatrix	0.05	0.04	0.05	0.05	0.05
occur	0.04	0.05	0.06	0.06	0.06
peephole	0.38	0.49	0.63	0.49	0.92
progeom	0.08	0.09	0.10	0.10	0.12
qplan	0.94	1.28	1.61	1.59	2.23
qsortapp	0.04	0.05	0.05	0.06	0.06
query	0.07	0.08	0.07	0.07	0.07
rdtok	0.46	0.61	0.70	0.87	1.05
read	0.46	0.95	1.12	1.08	1.19
serialize	0.07	0.07	0.09	0.11	0.37
tak	0.06	0.06	0.09	0.08	0.06
warplan	0.30	0.50	0.44	0.55	1.32
zebra	1.36	0.70	34.05	3.82	3.03
Average	0.26	0.32	1.88	0.52	0.73

Table 3: Annotation Times: LP

provides an idea of the overhead introduced in the program. The results for this evaluation are shown in tables 4 and 5.

7.4 Discussion

Regarding the analysis times, for most benchmarks the comparison between the Def and ShFr analyzers reflect the relative complexity of those two analyzers: the ShFr analyzer not only abstracts groundness information (and groundness dependencies) but also sharing and freeness. As a result its abstract operations are more complex than those of Def. The few cases in which Def does not compare well (e.g., `aiakl`) are due to loss of accuracy. The case of the Free analyzer is different. The information abstracted by Free is also less complex than that of ShFr. Also its solved form equips it with a sort of widening that allows it to keep a compact form, even when no information is known regarding the program variables (note that while the "top" abstraction for ShFr is the powerset of the set of variables involved, for Free is just the set of singletons). However, the Free analyser is in most cases slightly more expensive than ShFr and, in a few cases (e.g., `deriv`, `aiakl`, `qplan`, etc.), significantly more expensive. The cause is the lack of groundness information which makes its abstractions larger, the abstract

Bench. Program	Total CGEs					Uncond. CGEs				
	Local	Def	Free	FD	ShFr	Local	Def	Free	FD	ShFr
aiakl	2	2	2	2	2	0	0	0	0	2
ann	14	14	12	12	12	0	0	0	0	0
bid	6	6	5	5	5	0	3	0	5	5
boyer	2	2	2	2	2	0	0	0	0	0
browse	4	4	4	4	4	0	0	0	0	0
deriv	4	4	4	4	4	0	0	0	4	4
fib	1	1	1	1	1	1	1	1	1	1
grammar	0	0	0	0	0	0	0	0	0	0
hanoiapp	1	1	1	1	1	0	1	0	1	1
mmatrix	2	2	2	2	2	0	0	0	2	2
occur	2	2	2	2	2	0	1	0	2	2
peephole	2	2	2	2	2	0	1	0	1	1
progeom	2	2	1	1	1	0	1	0	1	1
qplan	20	20	18	18	18	0	11	0	16	16
qsortapp	1	1	1	1	1	0	1	0	1	1
query	2	2	1	1	1	0	0	1	1	1
rdtok	0	0	0	0	0	0	0	0	0	0
read	1	1	1	1	1	0	0	0	1	1
serialize	1	1	1	1	1	0	0	0	0	0
tak	1	1	1	1	1	1	1	1	1	1
warplan	9	9	9	9	9	0	0	0	0	0
zebra	2	2	2	2	1	1	1	1	1	1

Table 4: Parallel Expressions: LP

operations more expensive, and increases the number of different calling patterns. All this problems seem to be solved when Def and Free are combined.

When comparing the annotation times, the first surprise is the high time spent in the annotation of `zebra` when using the information provided by Free. The reason is the high number of free variables which definitely dependent on some other variable. Since Free cannot represent definite dependencies (as ShFr does), the translation of the information at each program point results in a very large number of implications. Handling this implications (e.g., performing their closure) is very expensive. However, if we forget this particular case, the average annotation times for Free is 0.48. Then, it can be said that the annotation times for each analyser clearly show the relative complexity of the information abstracted by the analysers, and therefore, the complexity of its transformation into the GI domain, and that of the operations handling such translations.

Finally, and regarding the effectiveness results, we can conclude that although the Def and Free analyzers are not very effective for the parallelization task by themselves, their combination results in a quite effective analysis: for most cases the parallelization using FD is as effective as that performed using ShFr. Only in three cases FD is less effective (`aiakl`, `boyer`, and `zebra`), due to the better dependency information provided by the set sharing implemented in the ShFr domain.

Bench. Program	Conditions: def/indep				
	Local	Def	Free	FD	ShFr
aiakl	0/10	0/10	0/5	0/5	0/0
ann	14/36	9/26	11/16	6/14	6/14
bid	7/12	2/7	4/2	0/0	0/0
boyer	4/2	4/2	4/2	4/2	4/1
browse	3/7	2/2	3/7	2/2	2/2
deriv	4/16	0/4	4/4	0/0	0/0
fib	0/0	0/0	0/0	0/0	0/0
grammar	0/0	0/0	0/0	0/0	0/0
hanoiapp	2/1	0/0	2/1	0/0	0/0
mmatrix	2/8	0/2	2/2	0/0	0/0
occur	2/5	0/1	2/2	0/0	0/0
peephole	3/4	1/4	3/2	1/2	1/2
progeom	2/2	1/0	0/2	0/0	0/0
qplan	13/57	5/7	8/33	2/1	2/1
qsortapp	0/1	0/0	0/1	0/0	0/0
query	1/4	1/4	0/0	0/0	0/0
rdtok	0/0	0/0	0/0	0/0	0/0
read	1/6	0/6	1/0	0/0	0/0
serialize	0/4	0/4	0/1	0/1	0/1
tak	0/0	0/0	0/0	0/0	0/0
warplan	12/11	12/9	12/8	12/7	12/7
zebra	0/250	0/50	0/92	0/11	0/0

Table 5: Conditional Checks: LP

8 Experimental Results: Constraint Logic Programs

In this section we evaluate the efficiency and accuracy of the Def, Free, and FD analyzers when dealing with constraint logic programs.

8.1 Benchmarks

The set of benchmarks used includes programs in the set of examples of the CLP(R) distribution, programs designed for PrologIII and translated into CLP(R), and programs designed for CLP(R). Table 6 describes the benchmarks by the same parameters used in the description of the traditional logic programming benchmarks.

8.2 Efficiency and Effectiveness

Tables 7, 8, 9, and 10 present, respectively, the analysis times in seconds, the annotation times in seconds, the number of CGEs which actually result in parallelism and the number of these which are unconditional, and the number of definiteness and independence tests in the remaining CGEs. The parameters shown in each table are

Bench.	AgV	MV	Cl	Ls	Ps	Non	Sim	Mut	Gs
amp	9.27	30	59	306	23	61	39	0	26
bridge	7.79	20	19	95	6	17	83	0	7
circuit	6.83	13	18	72	8	38	62	0	12
dnf	2.34	7	32	40	3	0	100	0	14
laplace	6.00	12	4	4	2	0	100	0	4
mining	2.93	18	43	73	21	48	52	0	30
mmatrix	5.33	9	6	20	3	0	100	0	3
mg_extend	6.90	14	10	42	6	67	33	0	13
num	2.46	12	99	190	18	94	6	0	40
pic	8.25	9	4	17	4	100	0	0	8
power	3.43	19	42	75	18	50	50	0	28
runge_kutta	6.20	9	5	11	4	75	25	0	6
trapezoid	6.40	9	5	9	4	75	25	0	11

Table 6: Benchmark Profiles: CLP

identical to that used in the corresponding table of the previous section. The symbol "–" appearing in some of the Free data indicates that the analyzer does not finish in a reasonable time (one hour) the analysis of the associated benchmark.

Benchmark	Def	Free	FD
amp	3.08	–	9.00
bridge	0.14	–	0.46
circuit	0.30	4.79	0.92
dnf	0.43	2.38	1.63
laplace	0.03	–	0.07
mining	0.34	3.53	3.63
mmatrix	0.03	0.10	0.06
mg_extend	0.18	0.25	0.55
num	0.57	2.03	2.41
pic	0.06	0.10	0.18
power	1.14	5.45	2.63
runge_kutta	0.02	0.16	0.07
trapezoid	0.84	0.23	1.22
Average	0.53	2.43	2.02

Table 7: Analysis Times: CLP

8.3 Discussion

The first thing that can be noticed, regarding the analysis efficiency, is that, despite the fact that most benchmarks are relatively small, the analysis times are quite high, specially for the Free analyzer (and sometimes also FD). It can be argued that in CLP programs the number of clause variables is usually very high, specially when compared to that of LP programs, even for small programs. This significantly affects

Benchmark	Def	Free	FD
amp	3.89	–	6.96
bridge	0.59	–	0.79
circuit	0.45	0.44	0.55
dnf	0.18	0.30	0.25
laplace	0.03	–	0.03
mining	0.42	0.46	0.55
mmatrix	0.10	0.14	0.11
mg_extend	0.38	0.43	0.51
num	1.11	1.49	1.58
pic	0.14	0.15	0.14
power	0.55	1.55	1.75
runge_kutta	0.07	0.09	0.06
trapezoid	0.06	0.04	0.07
Average	0.57	0.93	0.96

Table 8: Annotation Times: CLP

the Free analyzer, yielding very low performance. Performance is sometimes improved when combined with Def, i.e. in FD. Regarding the effectiveness of the information in the parallelization task, it is clear that the information provided by FD improves over the other two in a good number of cases. This fact, in combination with the performance results, allows to conclude that, as in the evaluation of traditional logic programs, the Def and Free analyzers are not very effective on their own, but their effectiveness is significantly improved when they are combined. However, when talking of CLP programs, this conclusion has to be carefully considered. Note that, even after the combination, the number of resulting run-time tests in the parallel programs are excessive. This can possibly reduce the effectiveness of the parallelizations.

There are several causes for these problems. On the one hand, the high number of variables usually involved, the complexity of the constraints abstracted, and the need to take into account the propagation of information through the bindings between Herbrand and numerical variables, significantly increase the complexity of both the abstractions and the abstract operations. On the other hand, more accurate domains usually also imply domains which are more complex.

It also has to be taken into account that none of the domains evaluated were defined for abstracting the information needed for an effective parallelization. While definiteness information (Def) is certainly vital, previous evaluations [BdlBH94b] have shown the importance of freeness information in order to avoid parallelizing expressions whose conditions can never succeed. Also, the independence information abstracted by Free is not enough for the purposes of reducing checks, since as soon as a variable becomes possibly constrained during the analysis, all independence information for such variable is lost. The key point in CLP, in order to keep track of the required properties, is then to accurately abstract the constraint solving behaviour.

Bench. Program	Total CGEs			Uncond. CGEs		
	Def	Free	FD	Def	Free	FD
amp	5	–	5	0	–	0
bridge	0	–	0	0	–	0
circuit	3	2	2	0	0	0
dnf	14	14	14	12	0	12
laplace	1	–	1	1	–	1
mining	5	4	4	1	0	2
mmatrix	2	2	2	0	0	0
mg_extend	0	0	0	0	0	0
num	16	16	16	5	10	10
pic	4	3	3	0	0	0
power	5	5	5	1	1	1
runge_kutta	2	1	1	0	0	0
trapezoid	1	1	1	0	0	0

Table 9: Parallel Expressions: CLP

Bench. Program	Conditions: def/indep		
	Def	Free	FD
amp	1/10	–	1/10
bridge	0/0	–	0/0
circuit	1/5	0/10	0/3
dnf	0/2	0/30	0/2
laplace	0/0	–	0/0
mining	3/5	5/5	2/4
mmatrix	0/2	2/8	0/2
mg_extend	0/0	0/0	0/0
num	0/24	0/20	0/19
pic	2/9	6/8	1/3
power	3/40	3/29	3/29
runge_kutta	5/0	6/0	3/0
trapezoid	0/9	0/9	0/9

Table 10: Conditional Checks: CLP

Recently, a new abstract domain (LSign) for CLP languages has been defined [MS94]. This domain is aimed at inferring accurate information about possible interaction between linear arithmetic equalities and inequalities. The key idea is to abstract the actual coefficients and constants in constraints by their "sign". A preliminary implementation of this domain shows very promising accuracy. Unfortunately, accuracy is paid in efficiency, specially when interactions among Herbrand and numerical constraints appear. Also, the existence of many different abstractions with identical concretization makes the implementation more difficult and slow. We would like to preserve such accuracy while improving the efficiency of the analyzer. In order to do this we are currently studying different normal forms which will allow us to define a unique element as the representative of the class of abstractions with the same concretization, without losing any accuracy. We are also studying methods to combine LP and CLP analysers in a way that they can accurately abstract the LP (Herbrand domain) and CLP (numerical or other domains) parts of a program, respectively, keeping the interaction minimal, and thus, allowing each abstract domain to focus on the concrete domain it was designed for.

9 Conclusions

We have proposed methods for automatically parallelizing CLP programs with the aid of program analysis, by transforming them to explicitly concurrent programs in our parallel CC platform *CIAO*. These techniques are also useful for performing similar transformations using AKL as a target, as shown in [BH95]. In doing this we have used the recent generalization of the concept of independence to the constraint logic programming (CLP) paradigm. Several abstract domains specifically designed for CLP languages, and whose information can be used to detect the generalized independence conditions which have been recently defined, have been studied. Our work shows that automatic compile-time parallelization of CLP programs via abstract interpretation and transformation is feasible. The analysis and transformation process is shown to be reasonably efficient and effective, specially considering that the domains used were not designed specifically for the parallelization application. Future work includes designing and studying other domains specifically for the task, as well as performing the parallelization at finer levels of granularity, using the "local independence" notions introduced in [BHMR94, Bue94].

References

[BdlBH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

[BdlBH94b] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BH95] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from CLP to AKL. Technical Report CLIP7/95.0, ACCLAIM Deliverable D3.3/3-A2, Facultad de Informática, UPM, June 1995.

[BHMR94] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[Bue94] F. Bueno. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation.* PhD thesis, Universidad Politécnica de Madrid (UPM), October 1994.

[Bue95] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, ACCLAIM Deliverable D3.2/3-A4, Facultad de Informática, UPM, June 1995.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[CDY91] M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming*, pages 79–96, Paris, France, June 1991. MIT Press.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In Springer-Verlag, editor, *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994.

[CH95] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. Technical Report CLIP14/95.0, ACCLAIM Deliverable D4.3/2-A2, Facultad de Informática, UPM, June 1995.

[CMB+93] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DJBC93] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, June 1993.

[dlBG94] María José García de la Banda García. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.

[dlBH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, Cambridge, MA, October 1993.

[dlBHB+95] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. In *ACM Transactions on Programming Languages and Systems.* ACM, 1995. To appear.

[dlBHM93] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[Her86] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HG91] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[HtCg94] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[HWD92] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[JL88] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming, to appear*, 1994.

[Kal87] L. Kale. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[Lin88] Y.-J. Lin. *A Parallel Implementation of Logic Programs*. PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[MH89a] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[MH89b] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90a] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH90b] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[MS92] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

[MS94] K. Marriott and P. Stuckey. Approximating Interaction Between Linear Arithmetic Constraints. In *1994 International Symposium on Logic Programming*, pages 571–585. MIT Press, 1994.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[She92] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[Son86] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[War90] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[WHD88] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.